Process Accessing Library (PAL): An Approach to Interprocess Communication

by Steven G. Betten

ARL-CR-470 **May 2001**

prepared by

Steven G. Betten
University of Maryland
College Park, MD 20742

under contract
DAAD17-00-P-0960

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-CR-470 **May 2001**

Process Accessing Library (PAL): An Approach to Interprocess Communication

Steven G. Betten

University of Maryland

Abstract

Interprocess communication is a topic of study in the high performance computing community because of its applications in runtime analysis and code coupling. Existing approaches to such communication include sockets, message passing, shared memory, and distributed shared memory. Proposed is a "process accessing" approach in which a program directly accesses desired data in the working memory of another program. This approach has its origins in debugger programs, which access the working memory of the program they are debugging. Major benefits of the process accessing model are that it provides access to computational results without pausing computations, it uses a minimal amount of memory, and it requires only trivial modifications to the computational code in order to access its working memory. The process accessing library (PAL) is an implementation of the process accessing approach.

Acknowledgments

I would like to acknowledge the suggestions and support of Jerry Clarke, Dr. Raju Namburu, Dr. Andrew Mark, Charles Nietubicz, Dr. N. Radhakrishnan, and the members of the Scientific Visualization Team, all of the U.S. Army Research Laboratory, Aberdeen Proving Ground, MD. Further, I would like to acknowledge the support of Professor Jeff Hollingsworth of the University of Maryland, College Park.

INTENTIONALLY LEFT BLANK.

Contents

INTENTIONALLY LEFT BLANK.

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

The U.S. Army Research Laboratory (ARL) is interested in the task of sharing data between executing programs. Applications of this concept include runtime analysis and code coupling. In runtime analysis, the goal is to provide the ability to analyze or visualize the progress of an executing computational code. The aim of code coupling is to combine two existing codes so that they cooperatively solve a problem by sharing results. Existing interprocess communication models for these applications include sockets, message passing, shared memory, and distributed shared memory. Proposed is a "process accessing" approach in which a program directly accesses the working memory of another program from which it desires data. The process accessing library (PAL) is an implementation of this last model. Major benefits of the process accessing model are that it provides access to computational results without pausing the code, it uses a minimal amount of memory, and it requires only trivial modifications to a computational code in order to access its working memory.

2. Approaches

Runtime analysis and code coupling are major fields of study in high performance computing. There are a couple of benefits of having runtime analysis capabilities. One benefit is the ability to steer the direction of a computation while it proceeds. Another benefit is the savings in processing time that one gains from stopping execution as soon as further computation becomes unnecessary. Code coupling has been gaining momentum as a technique for solving complex computational problems because using multiple codes to solve a problem allows each specialized code to focus on the aspect of the problem at which it excels. Existing approaches to runtime analysis and code coupling include sockets, message passing, shared memory, and distributed shared memory. PAL embodies the process accessing approach.

2.1 Sockets

A common way to share data between two programs is by sockets. Sockets are supported at the system level, and they allow fast, bidirectional, first-in first-out communication of raw data between programs running on a single computer or on multiple computers on a network. When using sockets, one must be weary of the potential of overrunning the buffer and corrupting data that is being transferred.

2.2 Message Passing

From the user point of view, message passing is similar to sockets except that message passing provides some features that are especially helpful for programming large-scale parallel programs. For example, message passing makes it easy to establish many connections at once, and it also always ensures that no data is corrupted during transfer between programs. Packages exist that integrate computational codes and runtime analyzers using message passing. One such package is pV3 [1], which utilizes the parallel virtual machine (PVM) [2] to allow runtime visualization. It is also possible to use message passing to couple certain computational codes. The message passing interface (MPI) standard provides the ability to execute multiple codes in the same communication sphere, allowing messages to travel between them [3]. There are also products that facilitate the coupling of certain MPI-based codes (e.g., reference [4]).

2.3 Shared Memory

The most intuitive way to share data between programs is to have them share memory. Support for sharing unstructured blocks of memory exists at the system level [5]. The system-level support includes simple locking mechanisms.

2.4 Distributed Shared Memory

Distributed shared memory software extends the shared memory model by giving the appearance of shared memory but actually spreading memory usage across multiple computers on a network and using sockets to communicate between the computers (e.g., references [6, 7]). The distributed interactive computing environment (DICE) is distributed shared memory software that supports runtime analysis by having the computational code periodically write its results to a section of shared memory called the DICE object directory (DOD) [8]. Analysis programs then access the results in the DOD. When they are coupled by way of DICE, computational codes exchange results via the DOD.

2.5 Process Accessing With PAL

PAL has applications in runtime analysis and code coupling. For runtime analysis, only the analysis program uses PAL and its accessing features, whereas in code coupling, both codes use PAL to access each other's data. Major benefits of the process accessing model, relative to other approaches, are that it provides access to computational results without needing to pause the code, it decreases the amount of memory used, and it requires minimal modifications to the computational code. It is best to use the process accessing model when both programs are running on the same computer.

PAL is written in C++ and provides many helpful object-oriented abstractions of system and programming concepts, such as processes, process groups, and global variables. The Appendix includes the listing of the example program PAD, which is written in C++ and uses PAL to access a parallel MPI program written in C. Typical usage of PAL by a program is as follows (accessor denotes the program that uses PAL; accessee denotes the program that PAL accesses).

(1) The accessor creates a process group, which represents the process or processes that are executing the accessee.

The user specifies the path of the accessee and whether it is a scalar program or an MPI-based parallel program. If there is more than one instance of the accessee running, the user provides a process id to uniquely identify the desired group. PAL creates handles to the process or processes in that group. Also, PAL scans the accessee code and stores the name and location of each C and C++ global variable and each Fortran77 and Fortran90 common block variable.

(2) Via the process group, the accessor acquires handles to each desired variable of the accessee.

If PAL fails to automatically create a handle to desired data, it is possible to manually create a variable by supplying a name and an offset.

(3) The accessor uses the variable handles to read and write accessee variable values.

PAL provides functions to access primitive types, such as integers, floating-point numbers, and characters, and to access contents of arrays and other pointer-based data structures. All of these functions require arguments specifying the accessee process to access, the data buffer to use, and the number of bytes to transfer; an offset into the accessee variable is optional. Data transfers occur directly between the working memory of the accessee and the working memory of the accessor.

3. Comparisons

3.1 General Application of Each Approach

Sockets are useful for simple, fast communication between two programs. The forte of message passing is its ease of programming and reliability for distributed, parallel programs. The shared memory model is generally the most intuitive model with which to program because it provides direct access to data that is uniform in every program, but it requires more work to maintain data integrity than with message passing. Distributed shared memory allows the use of the shared memory model on distributed systems. The process accessing model is best for scalar and parallel programs that run on a single machine and for which execution speed, memory usage, or minimal code modification are the principal concerns.

3.2 Encapsulation

One point of comparison is how much the accessor needs to know about the internal data format of the accessee (i.e., how well encapsulated the data of the accessee is). It can be difficult to synchronize advanced data structures, such as C++ classes, across multiple, asynchronous accesses with shared memory because shared memory libraries usually do not support encapsulation. An advantage of using message passing is that exchange between programs occurs via a standard interface, so message passing programs do not need to concern themselves with each other's internal data representations. MPI permits users to define and transfer data structures that are composites of primitive types, allowing programs to easily communicate complex data. Employing the process accessing model imposes the need of accessors to fully know the internal data representation of accessees.

3.3 Extensibility

Another consideration is extensibility, defined here as the ease with which one can couple an analysis or computational program to an existing code. Shared memory systems are generally extensible, as any program that supplies the correct key can access the shared memory that the code is using. With PAL, any program with the correct user id can access the working memory of the code. In both of these cases, it is usually a fairly straightforward matter to attach an analysis program because such attachments do not necessarily require any changes to the computational code. Coupling codes using shared memory would require adding synchronization to the computational code in order to maintain data integrity. The changes needed to couple codes using the process accessing model would depend on the type of coupling. It might be that only one code, the *consumer*, uses data from the other, the *producer*, in which case it would only be necessary to ensure that the producer provides global access to its computational results and to modify the consumer so that it acquires data by way of PAL. On the other hand, if both codes require data from each other, then both must make their computational data globally available and use PAL to access the data of the other code. Extension is more difficult with sockets and message passing, since the computational code would have to initiate the transfer of all data that the analysis program or other computational code would require. For the sake of extensibility, shared memory would probably be most preferable, with process accessing being just as viable with respect to runtime analysis.

3.4 Intermediate Storage: Data Format, Memory Usage, and Memory Access

In some situations, data format and memory usage can be concerns. A common procedure, one that DICE uses, is for computational codes to place a copy of their results into shared memory in a more-accessible format [8]. This procedure increases memory usage but facilitates data interchange with other programs. Another benefit of this technique is that accessors to code results will go to the copy of the code results and will not compete with accesses by the code to its working memory. PAL saves memory by

transferring data directly between the working memories of the accessor and accessee. However, working with the data in the working memory of the accessee could be frustrating if the internal data format of the accessee is complicated. Also, accesses by multiple programs to the working memory of the computational code might significantly inhibit the execution speed of the code. Using sockets incurs the memory overhead of buffers for transferring messages, without necessarily providing a common interface for data interchange. Message passing also requires buffer overhead, but at least all message passing programs use a common format for data exchange.

3.5 Computation Suspension: Data Integrity, Computation Duration, and Data Transfer Rates

There is a tradeoff in all the approaches to interprocess communication between data integrity and computation duration. For the purposes of this discussion, data is *integral* if it is all from the same iteration, and it is not integral otherwise. Computation duration is the time that a specific computational code takes to make all its computations for a given problem. The most straightforward way to ensure data integrity is to periodically suspend computations between iterations and to copy or send the data to another location from which other programs can read the data. On the other hand, if the code continues to compute as the accessor reads from it, the accessor data will likely be a mixture of results from more than one iteration. In many cases, the change in data between iterations is small enough that such an error is negligible and does not warrant any suspension of computations. One of the intended uses of PAL is to read data continually from executing code so that its users can acquire frequent and reasonably accurate updates without prolonging the computation duration. Sometimes, as in the case of DICE, transfers directly between the code and shared memory are faster than accesses via PAL. It appears that transfers via PAL are slower because PAL is unable to access the working memory of the code using *mmap* and *memcpy* as DICE does; instead, PAL accesses the computational working memory through the process file system using the less efficient *open*, *seek*, *read*, and *write* functions. The downside of stopping the code during computation is that it will take longer to carry out its computations. Using nonblocking, persistent communications minimizes time spent on communication for the message passing approach.

If minimizing computation duration and maximizing ease of data accessibility are both substantial concerns and data integrity and memory usage are not as important, then at least one hybrid approach exists. That is, have a PAL-based program continually extract data from the computational code, convert that data to a more accessible format, and store it in shared memory.

Table 1 contains data on average transfer rates between programs on an SGI Origin 2000 for the different approaches to sharing data between programs. The break at 4 MB is sometimes significant because the cache size of SGI Origin 2000 systems is 4 MB.

Table 1. Average transfer rates.

	Transfers ≤ 4 MB (MB/s)	Transfers > 4 MB (MB/s)
Shared Memory (NDGM)	400	120
Socket (ttcp)	140	140
Message Passing (MPI, persistent)	90	80
Process Accessing (PAL)	110	80

4. Conclusion

There are many tradeoffs in the use of PAL, and process accessing may not be the best approach in all cases. However, when minimizing computation duration, memory usage, or code modification are primary concerns, PAL is an attractive option.

5. References

1. Rice University. "pV3." http://www.ruf.rice.edu/~behr/pv3.html, 25 August 2000.
2. Oak Ridge National Laboratory. "PVM: Parallel Virtual Machine." http://www.epm.ornl.gov/pvm, 25 August 2000.
3. Argonne National Laboratory. "The Message Passing Interface (MPI) Standard." http://www-unix.mcs.anl.gov/mpi/index.html, 25 August 2000.
4. GMD German National Research Center for Information Technology. "MpCCI - Mesh-based Parallel Code Coupling Interface." http://www.mpcci.org, 25 August 2000.
5. SGI. "SGI TechPubs Library." http://techpubs.sgi.com/library, 25 August 2000.
6. Rice University. "The TreadMarks Distributed Shared Memory (DSM) System." http://www.cs.rice.edu/~willy/TreadMarks/overview.html, 25 August 2000.
7. Clarke, J. A. "Emulating Shared Memory to Simplify Distributed-Memory Programming." *IEEE Computational Science & Engineering*, vol. 4, no. 1, pp. 55–62, January–March 1997.
8. Clarke, J. A., C. E. Schmitt, and J. J. Hare. "Developing a Full Featured Application From an Existing Code Using the Distributed Interactive Computing Environment." DOD HPCMP Users Group Conference Proceedings, June 1998.

INTENTIONALLY LEFT BLANK.

Appendix. Listing of PAL

```
#include "PALProcess.h"
#include "PALProcessGroup.h"
#include "PALGlobalVar.h"
#include <iostream.h>

const long ROUNDS = 5;

struct simple_mpi_process {
  int rank;
  long data_size;
  int *data;
};

// Reads data from a group of running simple_mpi executables.
void main() {
  // 1. Create a PALProcessGroup.
  PALProcessGroup group(PA_GROUP_MPI, "/test/simple_mpi");

  // Some simple_mpi global vars:
  // int Grab_MPI_size;
  // int Grab_MPI_rank;
  // int *Grab_data_p;
  // long Grab_data_size;

  // Set the names of the desired global variables.
  char
    *MPI_size_name  = "Grab_MPI_size",
    *MPI_rank_name  = "Grab_MPI_rank",
    *data_name      = "Grab_data_p",
    *data_size_name = "Grab_data_size";


  // 2. Search for the desired global variables.
  const PALGlobalVar
    *MPI_size_var  = group.FindGlobalVar(MPI_size_name),
    *MPI_rank_var  = group.FindGlobalVar(MPI_rank_name),
    *data_var      = group.FindGlobalVar(data_name),
    *data_size_var = group.FindGlobalVar(data_size_name);
```

```
  // 3. If the program has found all the necessary global variables,
  // then get data from the processes.

  if ((MPI_size_var != 0) && (MPI_rank_var != 0) && (data_var != 0)
     && (data_size_var != 0))
    {
     // Get a handle to the MPI processes, in order to grab their data
     PALArrayList *processesList = group.GetProcesses();

     PALProcess **processes = (PALProcess **)processesList->ToArray();
     processes = (PALProcess **)processesList->ToArray();

     // Get the number of MPI processes.
     long MPI_size = processesList->Length();

     // Create an array to hold MPI_size simple_mpi_process structs.
     simple_mpi_process *simple_procs = new simple_mpi_process[MPI_size];

     for (long z = 0; z < MPI_size; z++) {
       MPI_rank_var->Read(processes[z], &simple_procs[z].rank,
                    sizeof(int));
       data_size_var->Read(processes[z], &simple_procs[z].data_size,
                     sizeof(long));
       simple_procs[z].data = new int[simple_procs[z].data_size];

       // Get data
       data_var->ReadIndirect(processes[z], simple_procs[z].data,
                       sizeof(int) * simple_procs[z].data_size);
     }
   }
}
```

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 2001	Final, June 2000–January 2001

4. TITLE AND SUBTITLE
Process Accessing Library (PAL): An Approach to Interprocess Communication

5. FUNDING NUMBERS
665803.731

6. AUTHOR(S)
Steven G. Betten*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
* University of Maryland, College Park, MD 20742

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-HC
Aberdeen Proving Ground, MD 21005-5067

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ARL-CR-470

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Interprocess communication is a topic of study in the high performance computing community because of its applications in runtime analysis and code coupling. Existing approaches to such communication include sockets, message passing, shared memory, and distributed shared memory. Proposed is a "process accessing" approach in which a program directly accesses desired data in the working memory of another program. This approach has its origins in debugger programs, which access the working memory of the program they are debugging. Major benefits of the process accessing model are that it provides access to computational results without pausing computations, it uses a minimal amount of memory, and it requires only trivial modifications to the computational code in order to access its working memory. The process accessing library (PAL) is an implementation of the process accessing approach.

14. SUBJECT TERMS	15. NUMBER OF PAGES
distributing computing, HPC, data modeling format	15
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

INTENTIONALLY LEFT BLANK.

NO. OF
COPIES ORGANIZATION

2 DEFENSE TECHNICAL
INFORMATION CENTER
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FT BELVOIR VA 22060-6218

1 COMMANDING GENERAL
US ARMY MATERIEL CMD
AMCRDA TF
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 INST FOR ADVNCD TCHNLGY
THE UNIV OF TEXAS AT AUSTIN
3925 W BRAKER LN STE 400
AUSTIN TX 78759-5316

1 US MILITARY ACADEMY
MATH SCI CTR EXCELLENCE
MADN MATH
THAYER HALL
WEST POINT NY 10996-1786

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL D
DR D SMITH
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS IS R
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI OK TL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS IS T
2800 POWDER MILL RD
ADELPHI MD 20783-1197

NO. OF
COPIES ORGANIZATION

ABERDEEN PROVING GROUND

2 DIR USARL
AMSRL CI LP (BLDG 305)

INTENTIONALLY LEFT BLANK.